Fifth Edition

Building Java Programs

A Back to Basics Approach

Stuart Reges
University of Washington

Marty Stepp
Stanford University

Pearson

SVP, Courseware Portfolio Management: Marcia Horton
Portfolio Manager: Matt Goldstein
Portfolio Manager Assistant: Meghan Jacoby
VP, Product Marketing: Roxanne McCarley
Director of Field Marketing: Tim Galligan
Product Marketing Manager: Yvonne Vannatta
Field Marketing Manager: Demetrius Hall
Marketing Assistant: Jon Bryant
Managing Content Producer: Scott Disanno
VP, Production & Digital Studio: Ruth Berry
Project Manager: Lakeside Editorial Services L.L.C.
Senior Specialist, Program Planning and Support: Deidra Headlee
Cover Design: Jerilyn Bockorick
R&P Manager: Ben Ferrini
R&P Project Manager: Lav Kush Sharma/Integra Publishing Services, Inc.
Cover Art: Marcell Faber/Shutterstock
Full-Service Project Management: Integra Software Services Pvt. Ltd.
Composition: Integra Software Services Pvt. Ltd.
Printer/Binder: LSC Communications
Cover Printer: Phoenix Color
Text Font: Monotype

Library of Congress Cataloging-in-Publication Data
Names: Reges, Stuart, author. | Stepp, Martin, author.
Title: Building Java programs: a back to basics approach / Stuart Reges, University of Washington, Marty Stepp, Stanford University.
Description: Fifth edition. | Hoboken, New Jersey: Pearson, 2019. | Includes index.
Identifiers: LCCN 2018050748 | ISBN 9780135471944 | ISBN 013547194X
Subjects: LCSH: Java (Computer program language)
Classification: LCC QA76.73.J38 R447 2019 | DDC 005.13/3—dc23 LC record available at https://lccn.loc.gov/2018050748

1 19

Pearson

ISBN 10: 0-13-547194-X
ISBN 13: 978-0-13-547194- 4

MyLab Programming

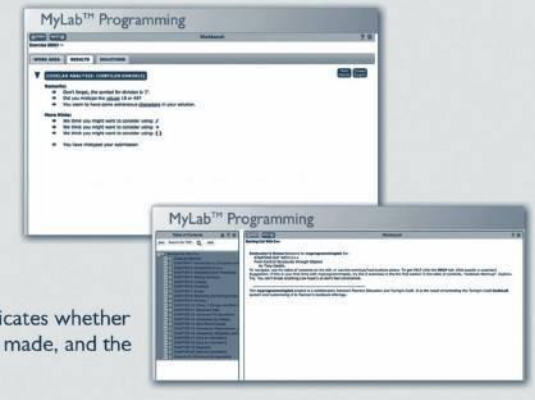
Through the power of practice and immediate personalized feedback, MyLab™ Programming helps students master programming fundamentals and build computational thinking skills.

PROGRAMMING PRACTICE

With MyLab Programming, your students will gain first-hand programming experience in an interactive online environment.

IMMEDIATE, PERSONALIZED FEEDBACK

MyLab Programming automatically detects errors in the logic and syntax of their code submission and offers targeted hints that enables students to figure out what went wrong and why.



GRADUATED COMPLEXITY

MyLab Programming breaks down programming concepts into short, understandable sequences of exercises. Within each sequence the level and sophistication of the exercises increase gradually but steadily.

DYNAMIC ROSTER

Students' submissions are stored in a roster that indicates whether the submission is correct, how many attempts were made, and the actual code submissions from each attempt.

PEARSON eTEXT

The Pearson eText gives students access to their textbook anytime, anywhere.

STEP-BY-STEP VIDEONOTE TUTORIALS

These step-by-step video tutorials enhance the programming concepts presented in select Pearson textbooks.

For more information and titles available with **MyLab Programming,** please visit **www.pearson.com/mylab/programming**

ALWAYS LEARNING

Pearson

Preface

The newly revised fifth edition of our *Building Java Programs* textbook is designed for use in a two-course introduction to computer science. We have class-tested it with thousands of undergraduates, most of whom were not computer science majors, in our CS1-CS2 sequence at the University of Washington. These courses are experiencing record enrollments, and other schools that have adopted our textbook report that students are succeeding with our approach.

Introductory computer science courses are often seen as "killer" courses with high failure rates. But as Douglas Adams says in *The Hitchhiker's Guide to the Galaxy*, "Don't panic." Students can master this material if they can learn it gradually. Our textbook uses a layered approach to introduce new syntax and concepts over multiple chapters.

Our textbook uses an "objects later" approach where programming fundamentals and procedural decomposition are taught before diving into object-oriented programming. We have championed this approach, which we sometimes call "back to basics," and have seen through years of experience that a broad range of scientists, engineers, and others can learn how to program in a procedural manner. Once we have built a solid foundation of procedural techniques, we turn to object-oriented programming. By the end of the course, students will have learned about both styles of programming.

The Java language is always evolving, and we have made it a point of focus in recent editions on newer features that have been added in Java 8 through 10. In the fourth edition we added a new Chapter 19 on Java's functional programming features introduced in Java 8. In this edition we integrate the JShell tool introduced in Java 9.

New to This Edition

The following are the major changes for our fifth edition:

- **JShell integration.** Java 9 introduced JShell, a utility with an interactive read-eval-print loop (REPL) that makes it easy to type Java expressions and immediately see their results. We find JShell to be a valuable learning tool that allows students to explore Java concepts without the overhead of creating a complete program. We introduce JShell in Chapter 2 and integrate JShell examples in each chapter throughout the text.
- **Improved Chapter 2 loop coverage.** We have added new sections and figures in Chapter 2 to help students understand `for` loops and create tables to find patterns in nested loops. This new content is based on our interactions with our own students as they solve programming problems with loops early in our courses.

- **Revamped case studies, examples, and other content.** We have rewritten or revised sections of various chapters based on student and instructor feedback. We have also rewritten the Chapter 10 (ArrayLists) case study with a new program focusing on elections and ranked choice voting.
- **Updated collection syntax and idioms.** Recent releases of Java have introduced new syntax and features related to collections, such as the <> "diamond operator;" collection interfaces such as `Lists`, `Sets`, and `Maps`; and new collection methods. We have updated our collection Chapters 10 and 11 to discuss these new features, and we use the diamond operator syntax with collections in the rest of the text.
- **Expanded self-checks and programming exercises.** With each new edition we add new programming exercises to the end of each chapter. There are roughly fifty total problems and exercises per chapter, all of which have been class-tested with real students and have solutions provided for instructors on our web site.
- **New programming projects.** Some chapters have received new programming projects, such as the Chapter 10 ranked choice ballot project.

Features from Prior Editions

The following features have been retained from previous editions:

- **Focus on problem solving.** Many textbooks focus on language details when they introduce new constructs. We focus instead on problem solving. What new problems can be solved with each construct? What pitfalls are novices likely to encounter along the way? What are the most common ways to use a new construct?
- **Emphasis on algorithmic thinking.** Our procedural approach allows us to emphasize algorithmic problem solving: breaking a large problem into smaller problems, using pseudocode to refine an algorithm, and grappling with the challenge of expressing a large program algorithmically.
- **Layered approach.** Programming in Java involves many concepts that are difficult to learn all at once. Teaching Java to a novice is like trying to build a house of cards. Each new card has to be placed carefully. If the process is rushed and you try to place too many cards at once, the entire structure collapses. We teach new concepts gradually, layer by layer, allowing students to expand their understanding at a manageable pace.
- **Case studies.** We end most chapters with a significant case study that shows students how to develop a complex program in stages and how to test it as it is being developed. This structure allows us to demonstrate each new programming construct in a rich context that can't be achieved with short code examples. Several of the case studies were expanded and improved in the second edition.
- **Utility as a CS1+CS2 textbook.** In recent editions, we added chapters that extend the coverage of the book to cover all of the topics from our second course in computer science, making the book usable for a two-course sequence. Chapters 12–19

explore recursion, searching and sorting, stacks and queues, collection implementation, linked lists, binary trees, hash tables, heaps, and more. Chapter 12 also received a section on recursive backtracking, a powerful technique for exploring a set of possibilities for solving problems such as 8 Queens and Sudoku.

This year also marks the release of our new *Building Python Programs* textbook, which brings our "back to basics" approach to the Python language. In recent years Python has seen a surge in popularity in introductory computer science classrooms. We have found that our materials and approach work as well in Python as they do in Java, and we are pleased to offer the choice of two languages to instructors and students.

Layers and Dependencies

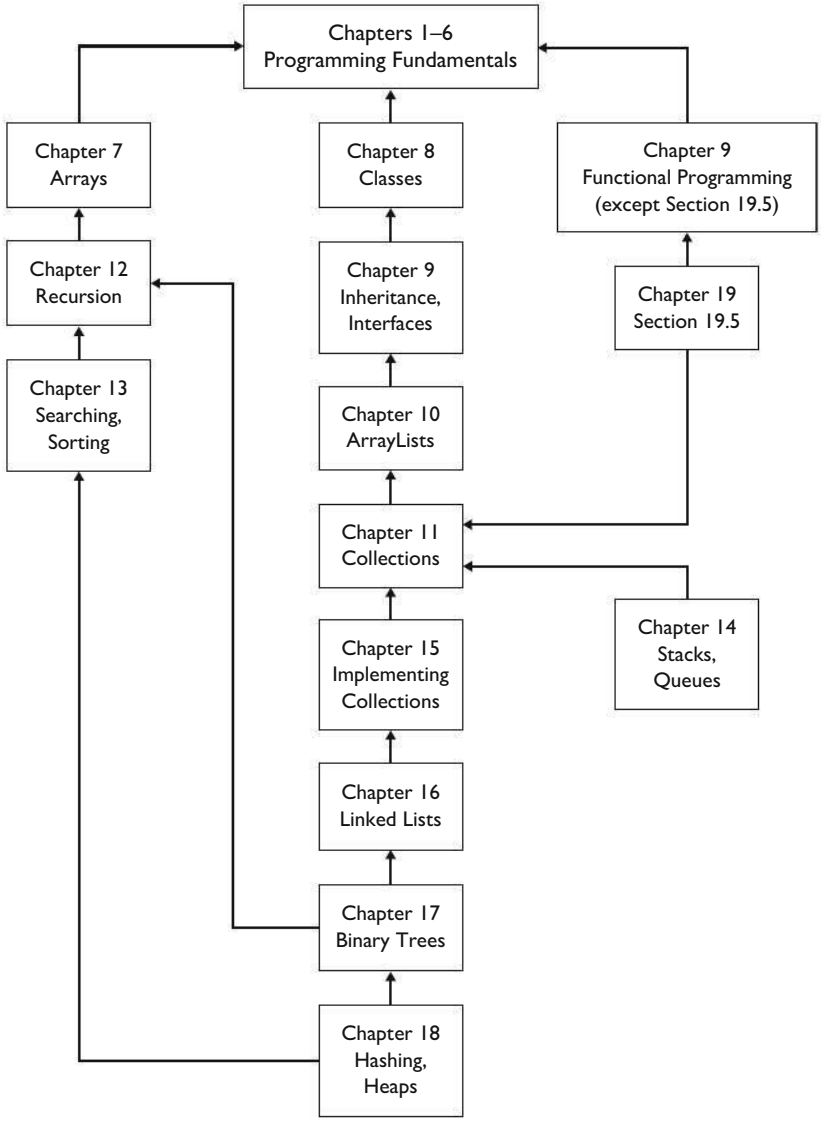
Many introductory computer science books are language-oriented, but the early chapters of our book are layered. For example, Java has many control structures (including for-loops, while-loops, and if/else-statements), and many books include all of these control structures in a single chapter. While that might make sense to someone who already knows how to program, it can be overwhelming for a novice who is learning how to program. We find that it is much more effective to spread these control structures into different chapters so that students learn one structure at a time rather than trying to learn them all at once.

The following table shows how the layered approach works in the first six chapters:

Chapter	Control Flow	Data	Programming Techniques	Input/Output
1	methods	`String` literals	procedural decomposition	`println`, `print`
2	definite loops (`for`)	variables, expressions, `int`, `double`	local variables, class constants, pseudocode	
3	return values	using objects	parameters	console input, 2D graphics (optional)
4	conditional (`if/else`)	`char`	pre/post conditions, throwing exceptions	`printf`
5	indefinite loops (`while`)	`boolean`	assertions, robust programs	
6		`Scanner`	token/line-based file processing	file I/O

Chapters 1–6 are designed to be worked through in order, with greater flexibility of study then beginning in Chapter 7. Chapter 6 may be skipped, although the case study in Chapter 7 involves reading from a file, a topic that is covered in Chapter 6.

The following is a dependency chart for the book:



Supplements

http://www.buildingjavaprograms.com/

Answers to all self-check problems appear on our web site and are accessible to anyone. Our web site has the following additional resources for students:

- **Online-only supplemental chapters**, such as a chapter on creating Graphical User Interfaces

- **Source code and data files** for all case studies and other complete program examples
- The **`DrawingPanel` class** used in the optional graphics Supplement 3G

Our web site has the following additional resources for teachers:

- **PowerPoint slides** suitable for lectures
- **Solutions** to exercises and programming projects, along with homework specification documents for many projects
- **Sample exams** and solution keys
- **Additional lab exercises** and **programming exercises** with solution keys
- **Closed lab creation tools** to produce lab handouts with the instructor's choice of problems integrated with the textbook

To access instructor resources, contact us at authors@buildingjavaprograms.com. The same materials are also available at http://www.pearsonhighered.com/cs-resources. To ask other questions related to resources, contact your Pearson sales representative.

MyLab Programming

MyLab Programming is an online practice and assessment tool that helps students fully grasp the logic, semantics, and syntax of programming. Through practice exercises and immediate, personalized feedback, MyLab Programming improves the programming competence of beginning students who often struggle with basic concepts and paradigms of popular high-level programming languages. A self-study and homework tool, the MyLab Programming course consists of hundreds of small practice exercises organized around the structure of this textbook. For students, the system automatically detects errors in the logic and syntax of code submissions and offers targeted hints that enable students to figure out what went wrong, and why. For instructors, a comprehensive grade book tracks correct and incorrect answers and stores the code inputted by students for review.

For a full demonstration, to see feedback from instructors and students, or to adopt MyLab Programming for your course, visit the following web site: www.pearson.com/mylab/programming

VideoNotes

VideoNote

We have recorded a series of instructional videos to accompany the textbook. They are available at the following web site: http://www.pearsonhighered.com/cs-resources

Roughly 3–4 videos are posted for each chapter. An icon in the margin of the page indicates when a VideoNote is available for a given topic. In each video, we spend 5–15 minutes walking through a particular concept or problem, talking about the challenges and methods necessary to solve it. These videos make a good supplement to the instruction given in lecture classes and in the textbook. Your new copy of the textbook has an access code that will allow you to view the videos.

Acknowledgments

First, we would like to thank the many colleagues, students, and teaching assistants who have used and commented on early drafts of this text. We could not have written this book without their input. Special thanks go to Hélène Martin, who pored over early versions of our first edition chapters to find errors and to identify rough patches that needed work. We would also like to thank instructor Benson Limketkai for spending many hours performing a technical proofread of the second edition.

Second, we would like to thank the talented pool of reviewers who guided us in the process of creating this textbook:

- Greg Anderson, Weber State University
- Delroy A. Brinkerhoff, Weber State University
- Ed Brunjes, Miramar Community College
- Tom Capaul, Eastern Washington University
- Tom Cortina, Carnegie Mellon University
- Charles Dierbach, Towson University
- H.E. Dunsmore, Purdue University
- Michael Eckmann, Skidmore College
- Mary Anne Egan, Siena College
- Leonard J. Garrett, Temple University
- Ahmad Ghafarian, North Georgia College & State University
- Raj Gill, Anne Arundel Community College
- Michael Hostetler, Park University
- David Hovemeyer, York College of Pennsylvania
- Chenglie Hu, Carroll College
- Philip Isenhour, Virginia Polytechnic Institute
- Andree Jacobson, University of New Mexico
- David C. Kamper, Sr., Northeastern Illinois University
- Simon G.M. Koo, University of San Diego
- Evan Korth, New York University

- Joan Krone, Denison University
- John H.E.F. Lasseter, Fairfield University
- Eric Matson, Wright State University
- Kathryn S. McKinley, University of Texas, Austin
- Jerry Mead, Bucknell University
- George Medelinskas, Northern Essex Community College
- John Neitzke, Truman State University
- Dale E. Parson, Kutztown University
- Richard E. Pattis, Carnegie Mellon University
- Frederick Pratter, Eastern Oregon University
- Roger Priebe, University of Texas, Austin
- Dehu Qi, Lamar University
- John Rager, Amherst College
- Amala V.S. Rajan, Middlesex University
- Craig Reinhart, California Lutheran University
- Mike Scott, University of Texas, Austin
- Alexa Sharp, Oberlin College
- Tom Stokke, University of North Dakota
- Leigh Ann Sudol, Fox Lane High School
- Ronald F. Taylor, Wright State University
- Andy Ray Terrel, University of Chicago
- Scott Thede, DePauw University
- Megan Thomas, California State University, Stanislaus
- Dwight Tuinstra, SUNY Potsdam
- Jeannie Turner, Sayre School
- Tammy VanDeGrift, University of Portland
- Thomas John VanDrunen, Wheaton College
- Neal R. Wagner, University of Texas, San Antonio
- Jiangping Wang, Webster University
- Yang Wang, Missouri State University
- Stephen Weiss, University of North Carolina at Chapel Hill
- Laurie Werner, Miami University
- Dianna Xu, Bryn Mawr College
- Carol Zander, University of Washington, Bothell

Finally, we would like to thank the great staff at Pearson who helped produce the book. Michelle Brown, Jeff Holcomb, Maurene Goo, Patty Mahtani, Nancy Kotary, and Kathleen Kenny did great work preparing the first edition. Our copy editors and the staff of Aptara Corp, including Heather Sisan, Brian Baker, Brendan Short, and Rachel Head, caught many errors and improved the quality of the writing. Marilyn Lloyd and Chelsea Bell served well as project manager and editorial assistant respectively on prior editions. For their help with the third edition we would like to thank Kayla Smith-Tarbox, Production Project Manager, and Jenah Blitz-Stoehr, Computer Science Editorial Assistant. Mohinder Singh and the staff at Aptara, Inc., were also very helpful in the final production of the third edition. For their great work on production of the fourth and fifth editions, we thank Louise Capulli and the staff of Lakeside Editorial Services, along with Carole Snyder at Pearson. Special thanks go to our lead editor at Pearson, Matt Goldstein, who has believed in the concept of our book from day one. We couldn't have finished this job without all of their hard work and support.

Stuart Reges
Marty Stepp

LOCATION OF VIDEO NOTES IN THE TEXT

http://www.pearson.com/cs-resources

VideoNote

Brief Contents

Contents

Chapter 1

Introduction to Java Programming

Introduction

This chapter begins with a review of some basic terminology about computers and computer programming. Many of these concepts will come up in later chapters, so it will be useful to review them before we start delving into the details of how to program in Java.

We will begin our exploration of Java by looking at simple programs that produce output. This discussion will allow us to explore many elements that are common to all Java programs, while working with programs that are fairly simple in structure.

After we have reviewed the basic elements of Java programs, we will explore the technique of procedural decomposition by learning how to break up a Java program into several methods. Using this technique, we can break up complex tasks into smaller subtasks that are easier to manage and we can avoid redundancy in our program solutions.

1.1 Basic Computing Concepts

Computers are pervasive in our daily lives, and, thanks to the Internet, they give us access to nearly limitless information. Some of this information is essential news, like the headlines on your favorite news web site. Computers let us share photos with our families and map directions to the nearest pizza place for dinner.

Lots of real-world problems are being solved by computers, some of which don't much resemble the one on your desk or lap. Computers allow us to sequence the human genome and search for DNA patterns within it. Computers in recently manufactured cars monitor each vehicle's status and motion, and computers are helping some cars to drive themselves. Digital music players and mobile devices such as Apple's iPhone actually have computers inside their small casings. Even the Roomba vacuum-cleaning robot houses a computer with complex instructions about how to dodge furniture while cleaning your floors.

But what makes a computer a computer? Is a calculator a computer? Is a human being with a paper and pencil a computer? The next several sections attempt to address this question while introducing some basic terminology that will help prepare you to study programming.

Why Programming?

At most universities, the first course in computer science is a programming course. Many computer scientists are bothered by this because it leaves people with the impression that computer science is programming. While it is true that many trained computer scientists spend time programming, there is a lot more to the discipline. So why do we study programming first?

A Stanford computer scientist named Don Knuth answers this question by saying that the common thread for most computer scientists is that we all in some way work with *algorithms*.

> **Algorithm**
>
> A step-by-step description of how to accomplish a task.

Knuth is an expert in algorithms, so he is naturally biased toward thinking of them as the center of computer science. Still, he claims that what is most important is not the algorithms themselves, but rather the thought process that computer scientists employ to develop them. According to Knuth,

> It has often been said that a person does not really understand something until after teaching it to someone else. Actually a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm.[1]

[1] Knuth, Don. *Selected Papers on Computer Science*. Stanford, CA: Center for the Study of Language and Information, 1996.

Knuth is describing a thought process that is common to most of computer science, which he refers to as *algorithmic thinking*. We study programming not because it is the most important aspect of computer science, but because it is the best way to explain the approach that computer scientists take to solving problems.

The concept of algorithms is helpful in understanding what a computer is and what computer science is all about. A major dictionary defines the word "computer" as "one that computes." Using that definition, all sorts of devices qualify as computers, including calculators, GPS navigation systems, and children's toys like the Furby. Prior to the invention of electronic computers, it was common to refer to humans as computers. The nineteenth-century mathematician Charles Peirce, for example, was originally hired to work for the U.S. government as an "Assistant Computer" because his job involved performing mathematical computations.

In a broad sense, then, the word "computer" can be applied to many devices. But when computer scientists refer to a computer, we are usually thinking of a universal computation device that can be programmed to execute any algorithm. Computer science, then, is the study of computational devices and the study of computation itself, including algorithms.

Algorithms are expressed as computer programs, and that is what this book is all about. But before we look at how to program, it will be useful to review some basic concepts about computers.

Hardware and Software

A computer is a machine that manipulates data and executes lists of instructions known as *programs*.

Program

A list of instructions to be carried out by a computer.

One key feature that differentiates a computer from a simpler machine like a calculator is its versatility. The same computer can perform many different tasks (playing games, computing income taxes, connecting to other computers around the world), depending on what program it is running at a given moment. A computer can run not only the programs that exist on it currently, but also new programs that haven't even been written yet.

The physical components that make up a computer are collectively called *hardware*. One of the most important pieces of hardware is the central processing unit, or *CPU*. The CPU is the "brain" of the computer: It is what executes the instructions. Also important is the computer's *memory* (often called random access memory, or *RAM*, because the computer can access any part of that memory at any time). The computer uses its memory to store programs that are being executed, along with their data. RAM is limited in size and does not retain its contents when the computer is turned off. Therefore, computers generally also use a *hard disk* as a larger permanent storage area.

Computer programs are collectively called *software*. The primary piece of software running on a computer is its operating system. An *operating system* provides an environment in which many programs may be run at the same time; it also provides a bridge between those programs, the hardware, and the *user* (the person using the computer). The programs that run inside the operating system are often called *applications*.

When the user selects a program for the operating system to run (e.g., by double-clicking the program's icon on the desktop), several things happen: The instructions for that program are loaded into the computer's memory from the hard disk, the operating system allocates memory for that program to use, and the instructions to run the program are fed from memory to the CPU and executed sequentially.

The Digital Realm

In the last section, we saw that a computer is a general-purpose device that can be programmed. You will often hear people refer to modern computers as *digital* computers because of the way they operate.

> **Digital**
>
> Based on numbers that increase in discrete increments, such as the integers 0, 1, 2, 3, etc.

Because computers are digital, everything that is stored on a computer is stored as a sequence of integers. This includes every program and every piece of data. An MP3 file, for example, is simply a long sequence of integers that stores audio information. Today we're used to digital music, digital pictures, and digital movies, but in the 1940s, when the first computers were built, the idea of storing complex data in integer form was fairly unusual.

Not only are computers digital, storing all information as integers, but they are also *binary*, which means they store integers as *binary numbers*.

> **Binary Number**
>
> A number composed of just 0s and 1s, also known as a base-2 number.

Humans generally work with *decimal* or base-10 numbers, which match our physiology (10 fingers and 10 toes). However, when we were designing the first computers, we wanted systems that would be easy to create and very reliable. It turned out to be simpler to build these systems on top of binary phenomena (e.g., a circuit being open or closed) rather than having 10 different states that would have to be distinguished from one another (e.g., 10 different voltage levels).

From a mathematical point of view, you can store things just as easily using binary numbers as you can using base-10 numbers. But since it is easier to construct a physical device that uses binary numbers, that's what computers use.

This does mean, however, that people who aren't used to computers find their conventions unfamiliar. As a result, it is worth spending a little time reviewing how binary

numbers work. To count with binary numbers, as with base-10 numbers, you start with 0 and count up, but you run out of digits much faster. So, counting in binary, you say

```
0
1
```

And already you've run out of digits. This is like reaching 9 when you count in base-10. After you run out of digits, you carry over to the next digit. So, the next two binary numbers are

```
10
11
```

And again, you've run out of digits. This is like reaching 99 in base-10. Again, you carry over to the next digit to form the three-digit number 100. In binary, whenever you see a series of ones, such as 111111, you know you're just one away from the digits all flipping to 0s with a 1 added in front, the same way that, in base-10, when you see a number like 999999, you know that you are one away from all those digits turning to 0s with a 1 added in front.

Table 1.1 shows how to count up to the base-10 number 8 using binary.

Table 1.1 Decimal vs. Binary

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

We can make several useful observations about binary numbers. Notice in the table that the binary numbers 1, 10, 100, and 1000 are all perfect powers of 2 (2^0, 2^1, 2^2, 2^3). In the same way that in base-10 we talk about a ones digit, tens digit, hundreds digit, and so on, we can think in binary of a ones digit, twos digit, fours digit, eights digit, sixteens digit, and so on.

Computer scientists quickly found themselves needing to refer to the sizes of different binary quantities, so they invented the term *bit* to refer to a single binary digit and the term *byte* to refer to 8 bits. To talk about large amounts of memory, they invented the terms "kilobytes" (KB), "megabytes" (MB), "gigabytes" (GB), and so on. Many people think that these correspond to the metric system, where "kilo" means 1000, but

that is only approximately true. We use the fact that 2^{10} is approximately equal to 1000 (it actually equals 1024). Table 1.2 shows some common units of memory storage:

Table 1.2 Units of Memory Storage

Measurement	Power of 2	Actual Value	Example
kilobyte (KB)	2^{10}	1024	500-word paper (3 KB)
megabyte (MB)	2^{20}	1,048,576	typical book (1 MB) or song (5 MB)
gigabyte (GB)	2^{30}	1,073,741,824	typical movie (4.7 GB)
terabyte (TB)	2^{40}	1,099,511,627,776	20 million books in the Library of Congress (20 TB)
petabyte (PB)	2^{50}	1,125,899,906,842,624	10 billion digital photos (1.5 PB)

The Process of Programming

The word *code* describes program fragments ("these four lines of code") or the act of programming ("Let's code this into Java"). Once a program has been written, you can *execute* it.

Program Execution

The act of carrying out the instructions contained in a program.

The process of execution is often called *running*. This term can also be used as a verb ("When my program runs it does something strange") or as a noun ("The last run of my program produced these results").

A computer program is stored internally as a series of binary numbers known as the *machine language* of the computer. In the early days, programmers entered numbers like these directly into the computer. Obviously, this is a tedious and confusing way to program a computer, and we have invented all sorts of mechanisms to simplify this process.

Modern programmers write in what are known as high-level programming languages, such as Java. Such programs cannot be run directly on a computer: They first have to be translated into a different form by a special program known as a *compiler*.

Compiler

A program that translates a computer program written in one language into an equivalent program in another language (often, but not always, translating from a high-level language into machine language).

A compiler that translates directly into machine language creates a program that can be executed directly on the computer, known as an *executable*. We refer to such compilers as *native compilers* because they compile code to the lowest possible level (the native machine language of the computer).

This approach works well when you know exactly what computer you want to use to run your program. But what if you want to execute a program on many different computers? You'd need a compiler that generates different machine language output for each of them. The designers of Java decided to use a different approach. They cared a lot about their programs being able to run on many different computers, because they wanted to create a language that worked well for the Web.

Instead of compiling into machine language, Java programs compile into what are known as *Java bytecodes*. One set of bytecodes can execute on many different machines. These bytecodes represent an intermediate level: They aren't quite as high-level as Java or as low-level as machine language. In fact, they are the machine language of a theoretical computer known as the *Java Virtual Machine (JVM)*.

Java Virtual Machine

A theoretical computer whose machine language is the set of Java bytecodes.

A JVM isn't an actual machine, but it's similar to one. When we compile programs to this level, there isn't much work remaining to turn the Java bytecodes into actual machine instructions.

To actually execute a Java program, you need another program that will execute the Java bytecodes. Such programs are known generically as *Java runtimes*, and the standard environment distributed by Oracle Corporation is known as the *Java Runtime Environment (JRE)*.

Java Runtime

A program that executes compiled Java bytecodes.

Most people have Java runtimes on their computers, even if they don't know about them. For example, Apple's Mac OS X includes a Java runtime, and many Windows applications install a Java runtime.

Why Java?

When Sun Microsystems released Java in 1995, it published a document called a "white paper" describing its new programming language. Perhaps the key sentence from that paper is the following:

> Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.[2]

This sentence covers many of the reasons why Java is a good introductory programming language. For starters, Java is reasonably simple for beginners to learn, and it embraces object-oriented programming, a style of writing programs that has been shown to be very successful for creating large and complex software systems.

[2] http://www.oracle.com/technetwork/java/langenv-140151.html

Java also includes a large amount of prewritten software that programmers can utilize to enhance their programs. Such off-the-shelf software components are often called *libraries*. For example, if you wish to write a program that connects to a site on the Internet, Java contains a library to simplify the connection for you. Java contains libraries to draw graphical user interfaces (GUIs), retrieve data from databases, and perform complex mathematical computations, among many other things. These libraries collectively are called the *Java class libraries*.

> **Java Class Libraries**
>
> The collection of preexisting Java code that provides solutions to common programming problems.

The richness of the Java class libraries has been an extremely important factor in the rise of Java as a popular language. The Java class libraries in version 10 include over 6000 entries.

Another reason to use Java is that it has a vibrant programmer community. Extensive online documentation and tutorials are available to help programmers learn new skills. Many of these documents are written by Oracle, including an extensive reference to the Java class libraries called the *API Specification* (API stands for Application Programming Interface).

Java is extremely platform independent; unlike programs written in many other languages, the same Java program can be executed on many different operating systems, such as Windows, Linux, and Mac OS X.

Java is used extensively for both research and business applications, which means that a large number of programming jobs exist in the marketplace today for skilled Java programmers. A sample Google search for the phrase "Java jobs" returned around 816,000,000 hits at the time of this writing.

The Java Programming Environment

You must become familiar with your computer setup before you start programming. Each computer provides a different environment for program development, but there are some common elements that deserve comment. No matter what environment you use, you will follow the same basic three steps:

1. Type in a program as a Java class.
2. Compile the program file.
3. Run the compiled version of the program.

The basic unit of storage on most computers is a *file*. Every file has a name. A file name ends with an *extension*, which is the part of a file's name that follows the period. A file's extension indicates the type of data contained in the file. For example, files with the extension `.doc` are Microsoft Word documents, and files with the extension `.mp3` are MP3 audio files.

The Java program files that you create must use the extension `.java`. When you compile a Java program, the resulting Java bytecodes are stored in a file with the same name and the extension `.class`.

Most Java programmers use what are known as Integrated Development Environments, or IDEs, which provide an all-in-one environment for creating, editing, compiling, and executing program files. Some of the more popular choices for introductory computer science classes are Eclipse, IntelliJ, NetBeans, jGRASP, DrJava, BlueJ, and TextPad. Your instructor will tell you what environment you should use.

Try typing the following simple program in your IDE (the line numbers are not part of the program but are used as an aid):

```
1  public class Hello {
2      public static void main(String[] args) {
3          System.out.println("Hello, world!");
4      }
5  }
```

Don't worry about the details of this program right now. We will explore those in the next section.

Once you have created your program file, move to step 2 and compile it. The command to compile will be different in each development environment, but the process is the same (typical commands are "compile" or "build"). If any errors are reported, go back to the editor, fix them, and try to compile the program again. (We'll discuss errors in more detail later in this chapter.)

Once you have successfully compiled your program, you are ready to move to step 3, running the program. Again, the command to do this will differ from one environment to the next, but the process is similar (the typical command is "run"). The diagram in Figure 1.1 summarizes the steps you would follow in creating a program called `Hello.java`.

In some IDEs, the first two steps are combined. In these environments the process of compiling is more incremental; the compiler will warn you about errors as you type in code. It is generally not necessary to formally ask such an environment to compile your program because it is compiling as you type.

When your program is executed, it will typically interact with the user in some way. The `Hello.java` program involves an onscreen window known as the *console*.

Console Window

A special text-only window in which Java programs interact with the user.

The console window is a classic interaction mechanism wherein the computer displays text on the screen and sometimes waits for the user to type responses. This is known as *console* or *terminal interaction*. The text the computer prints to the console window is known as the *output* of the program. Anything typed by the user is known as the console *input*.

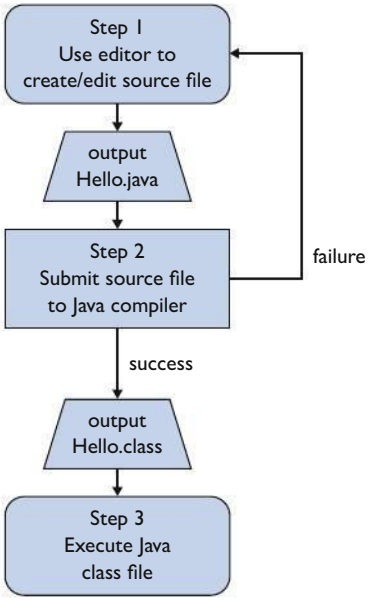


Figure 1.1 Creation and execution of a Java program

To keep things simple, most of the sample programs in this book involve console interaction. Keeping the interaction simple will allow you to focus your attention and effort on other aspects of programming.

1.2 And Now—Java

It's time to look at a complete Java program. In the Java programming language, nothing can exist outside of a *class*.

> **Class**
>
> A unit of code that is the basic building block of Java programs.

The notion of a class is much richer than this, as you'll see when we get to Chapter 8, but for now all you need to know is that each of your Java programs will be stored in a class.

It is a tradition in computer science that when you describe a new programming language, you should start with a program that produces a single line of output with the words, "Hello, world!" The "hello world" tradition has been broken by many authors of Java books because the program turns out not to be as short and simple when it is written in Java as when it is written in other languages, but we'll use it here anyway.

Here is our "hello world" program:

```
1  public class Hello {
2      public static void main(String[] args) {
3          System.out.println("Hello, world!");
4      }
5  }
```

This program defines a class called `Hello`. Oracle has established the convention that class names always begin with a capital letter, which makes it easy to recognize them. Java requires that the class name and the file name match, so this program must be stored in a file called `Hello.java`. You don't have to understand all the details of this program just yet, but you do need to understand the basic structure.

The basic form of a Java class is as follows:

```
public class <name> {
    <method>
    <method>
    ...
    <method>
}
```

This type of description is known as a *syntax template* because it describes the basic form of a Java construct. Java has rules that determine its legal *syntax* or grammar. Each time we introduce a new element of Java, we'll begin by looking at its syntax template. By convention, we use the less-than (<) and greater-than (>) characters in a syntax template to indicate items that need to be filled in (in this case, the name of the class and the methods). When we write "..." in a list of elements, we're indicating that any number of those elements may be included.

The first line of the class is known as the *class header*. The word `public` in the header indicates that this class is available to anyone to use. Notice that the program code in a class is enclosed in curly brace characters (`{ }`). These characters are used in Java to group together related bits of code. In this case, the curly braces are indicating that everything defined within them is part of this public class.

So what exactly can appear inside the curly braces? What can be contained in a class? All sorts of things, but for now, we'll limit ourselves to *methods*. Methods are the next-smallest unit of code in Java, after classes. A method represents a single action or calculation to be performed.

Method

A program unit that represents a particular action or computation.

Simple methods are like verbs: They command the computer to perform some action. Inside the curly braces for a class, you can define several different methods.

At a minimum, a complete program requires a special method that is known as the `main` method. It has the following syntax:

```
public static void main(String[] args) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

Just as the first line of a class is known as a class header, the first line of a method is known as a *method header*. The header for `main` is rather complicated. Most people memorize this as a kind of magical incantation. You want to open the door to Ali Baba's cave? You say, "Open Sesame!" You want to create an executable Java program? You say, `public static void main(String[] args)`. A group of Java teachers make fun of this with a website called publicstaticvoidmain.com.

Just memorizing magical incantations is never satisfying, especially for computer scientists who like to know everything that is going on in their programs. But this is a place where Java shows its ugly side, and you'll just have to live with it. New programmers, like new drivers, must learn to use something complex without fully understanding how it works. Fortunately, by the time you finish this book, you'll understand every part of the incantation.

Notice that the `main` method has a set of curly braces of its own. They are again used for grouping, indicating that everything that appears between them is part of the `main` method. The lines in between the curly braces specify the series of actions the computer should perform when it executes the method. We refer to these as the *statements* of the method. Just as you put together an essay by stringing together complete sentences, you put together a method by stringing together statements.

Statement

An executable snippet of code that represents a complete command.

Each statement is terminated by a semicolon. The sample "hello world" program has just a single statement that is known as a `println` statement:

```
System.out.println("Hello, world!");
```

Notice that this statement ends with a semicolon. The semicolon has a special status in Java; it is used to terminate statements in the same way that periods terminate sentences in English.

In the basic "hello world" program there is just a single command to produce a line of output, but consider the following variation (called `Hello2`), which has four lines of code to be executed in the `main` method:

```
1  public class Hello2 {
2      public static void main(String[] args) {
3          System.out.println("Hello, world!");
4          System.out.println();
5          System.out.println("This program produces four");
6          System.out.println("lines of output.");
7      }
8  }
```

Notice that there are four semicolons in the `main` method, one at the end of each of the four `println` statements. The statements are executed in the order in which they appear, from first to last, so the `Hello2` program produces the following output:

```
Hello, world!

This program produces four
lines of output.
```

Let's summarize the different levels we just looked at:

- A Java program is stored in a class.
- Within the class, there are methods. At a minimum, a complete program requires a special method called `main`.
- Inside a method like `main`, there is a series of statements, each of which represents a single command for the computer to execute.

It may seem odd to put the opening curly brace at the end of a line rather than on a line by itself. Some people would use this style of indentation for the program instead:

```
1  public class Hello3
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Hello, world!");
6      }
7  }
```

Different people will make different choices about the placement of curly braces. The style we use follows Oracle's official Java coding conventions, but the other style has its advocates too. Often people will passionately argue that one way is much better than the other, but it's really a matter of personal taste because each choice has some advantages and some disadvantages. Your instructor may require a particular style; if not, you should choose a style that you are comfortable with and then use it consistently.

Now that you've seen an overview of the structure, let's examine some of the details of Java programs.

Did You Know?

Hello, World!

The "hello world" tradition was started by Brian Kernighan and Dennis Ritchie. Ritchie invented a programming language known as C in the 1970s and, together with Kernighan, coauthored the first book describing C, published in 1978. The first complete program in their book was a "hello world" program. Kernighan and Ritchie, as well as their book *The C Programming Language*, have been affectionately referred to as "K & R" ever since.

Many major programming languages have borrowed the basic C syntax as a way to leverage the popularity of C and to encourage programmers to switch to it. The languages C++ and Java both borrow a great deal of their core syntax from C.

Kernighan and Ritchie also had a distinctive style for the placement of curly braces and the indentation of programs that has become known as "K & R style." This is the style that Oracle recommends and that we use in this book.

String Literals (Strings)

When you are writing Java programs (such as the preceding "hello world" program), you'll often want to include some literal text to send to the console window as output. Programmers have traditionally referred to such text as a *string* because it is composed of a sequence of characters that we string together. The Java language specification uses the term *string literals*.

In Java you specify a string literal by surrounding the literal text in quotation marks, as in

```
"This is a bunch of text surrounded by quotation marks."
```

You must use double quotation marks, not single quotation marks. The following is not a valid string literal:

```
'Bad stuff here.'
```

The following is a valid string literal:

```
"This is a string even with 'these' quotes inside."
```

String literals must not span more than one line of a program. The following is not a valid string literal:

```
"This is really
bad stuff
right here."
```

System.out.println

As you have seen, the `main` method of a Java program contains a series of statements for the computer to carry out. They are executed sequentially, starting with the first statement, then the second, then the third, and so on until the final statement has been executed. One of the simplest and most common statements is `System.out.println`, which is used to produce a line of output. This is another "magical incantation" that you should commit to memory. As of this writing, Google lists around 8,000,000 web pages that mention `System.out.println`. The key thing to remember about this statement is that it's used to produce a line of output that is sent to the console window.

The simplest form of the `println` statement has nothing inside its parentheses and produces a blank line of output:

```
System.out.println();
```

You need to include the parentheses even if you don't have anything to put inside them. Notice the semicolon at the end of the line. All statements in Java must be terminated with a semicolon.

More often, however, you use `println` to output a line of text:

```
System.out.println("This line uses the println method.");
```

The above statement commands the computer to produce the following line of output:

```
This line uses the println method.
```

Each `println` statement produces a different line of output. For example, consider the following three statements:

```
System.out.println("This is the first line of output.");
System.out.println();
System.out.println("This is the third, below a blank line.");
```

Executing these statements produces the following three lines of output (the second line is blank):

```
This is the first line of output.

This is the third, below a blank line.
```

Escape Sequences

Any system that involves quoting text will lead you to certain difficult situations. For example, string literals are contained inside quotation marks, so how can you include a quotation mark inside a string literal? String literals also aren't allowed to break across lines, so how can you include a line break inside a string literal?

The solution is to embed what are known as *escape sequences* in the string literals. Escape sequences are two-character sequences that are used to represent special characters. They all begin with the backslash character (\). Table 1.3 lists some of the more common escape sequences.

Table 1.3 Common Escape Sequences

Sequence	Represents
`\t`	tab character
`\n`	newline character
`\"`	quotation mark
`\\`	backslash character

Keep in mind that each of these two-character sequences actually stands for just a single character. For example, consider the following statement:

```
System.out.println("What \"characters\" does this \\ print?");
```

If you executed this statement, you would get the following output:

```
What "characters" does this \ print?
```

The string literal in the `println` has three escape sequences, each of which is two characters long and produces a single character of output.

While string literals themselves cannot span multiple lines (that is, you cannot use a carriage return within a string literal to force a line break), you can use the `\n` escape sequence to embed newline characters in a string. This leads to the odd situation where a single `println` statement can produce more than one line of output.

For example, consider this statement:

```
System.out.println("This\nproduces 3 lines\nof output.");
```

If you execute it, you will get the following output:

```
This
produces 3 lines
of output.
```

The `println` itself produces one line of output, but the string literal contains two newline characters that cause it to be broken up into a total of three lines of output. To produce the same output without new line characters, you would have to issue three separate `println` statements.

This is another programming habit that tends to vary according to taste. Some people (including the authors) find it hard to read string literals that contain `\n` escape sequences, but other people prefer to write fewer lines of code. Once again, you should make up your own mind about when to use the new line escape sequence.

`print` versus `println`

Java has a variation of the `println` command called `print` that allows you to produce output on the current line without going to a new line of output. The `println` command really does two different things: It sends output to the current line, and then it moves to the beginning of a new line. The `print` command does only the first of these. Thus, a series of `print` commands will generate output all on the same line. Only a `println` command will cause the current line to be completed and a new line to be started. For example, consider these six statements:

```
System.out.print("To be ");
System.out.print("or not to be.");
System.out.print("That is ");
System.out.println("the question.");
System.out.print("This is");
System.out.println(" for the whole family!");
```

These statements produce two lines of output. Remember that every `println` statement produces exactly one line of output; because there are two `println` statements here, there are two lines of output. After the first statement executes, the current line looks like this:

```
To be
     ^
```

The arrow below the output line indicates the position where output will be sent next. We can simplify our discussion if we refer to the arrow as the *output cursor*. Notice that the output cursor is at the end of this line and that it appears after a space. The reason is that the command was a `print` (doesn't go to a new line) and the string literal in the `print` ended with a space. Java will not insert a space for you unless you specifically request it. After the next `print`, the line looks like this:

```
To be or not to be.
                   ^
```

There's no space at the end now because the string literal in the second `print` command ends in a period, not a space. After the next `print`, the line looks like this:

```
To be or not to be.That is
                          ^
```

There is no space between the period and the word "That" because there was no space in the `print` commands, but there is a space at the end of the string literal in the third statement. After the next statement executes, the output looks like this:

```
To be or not to be.That is the question.

^
```

Because this fourth statement is a `println` command, it finishes the output line and positions the cursor at the beginning of the second line. The next statement is another `print` that produces this:

```
To be or not to be.That is the question.
This is ^
```

The final `println` completes the second line and positions the output cursor at the beginning of a new line:

```
To be or not to be.That is the question.
This is for the whole family!

^
```

These six statements are equivalent to the following two single statements:

```
System.out.println("To be or not to be.That is the question.");
System.out.println("This is for the whole family!");
```

Using the `print` and `println` commands together to produce lines like these may seem a bit silly, but you will see that there are more interesting applications of `print` in the next chapter.

Remember that it is possible to have an empty `println` command:

```
System.out.println();
```

Because there is nothing inside the parentheses to be written to the output line, this command positions the output cursor at the beginning of the next line. If there are `print` commands before this empty `println`, it finishes out the line made by those `print` commands. If there are no previous `print` commands, it produces a blank line. An empty `print` command is meaningless and illegal.

Identifiers and Keywords

The words used to name parts of a Java program are called *identifiers*.

> **Identifier**
>
> A name given to an entity in a program, such as a class or method.

Identifiers must start with a letter, which can be followed by any number of letters or digits. The following are all legal identifiers:

```
first          hiThere       numStudents       TwoBy4
```

The Java language specification defines the set of letters to include the underscore and dollar-sign characters (_ and $), which means that the following are legal identifiers as well:

```
two_plus_two      _count        $2donuts       MAX_COUNT
```

The following are illegal identifiers:

```
two+two           hi there      hi-There       2by4
```

Java has conventions for capitalization that are followed fairly consistently by programmers. All class names should begin with a capital letter, as with the `Hello`, `Hello2`, and `Hello3` classes introduced earlier. The names of methods should begin with lowercase letters, as in the `main` method. When you are putting several words together to form a class or method name, capitalize the first letter of each word after the first. In the next chapter we'll discuss constants, which have yet another capitalization scheme, with all letters in uppercase and words separated by underscores. These different schemes might seem like tedious constraints, but using consistent capitalization in your code allows the reader to quickly identify the various code elements.

For example, suppose that you were going to put together the words "all my children" into an identifier. The result would be:

- `AllMyChildren` for a class name (each word starts with a capital)
- `allMyChildren` for a method name (starts with a lowercase letter, subsequent words capitalized)
- `ALL_MY_CHILDREN` for a constant name (all uppercase, with words separated by underscores; described in Chapter 2)

Java is case sensitive, so the identifiers `class`, `Class`, `CLASS`, and `cLASs` are all considered different. Keep this in mind as you read error messages from the compiler. People are good at understanding what you write, even if you misspell words or make little mistakes like changing the capitalization of a word. However, mistakes like these cause the Java compiler to become hopelessly confused.

Don't hesitate to use long identifiers. The more descriptive your names are, the easier it will be for people (including you) to read your programs. Descriptive identifiers are worth the time they take to type. Java's `String` class, for example, has a method called `compareToIgnoreCase`.

Be aware, however, that Java has a set of predefined identifiers called *keywords* that are reserved for particular uses. As you read this book, you will learn many of these keywords and their uses. You can only use keywords for their intended purposes. You must be careful to avoid using these words in the names of identifiers. For example, if you name a method `short` or `try`, this will cause a problem, because `short` and `try` are reserved keywords. Table 1.4 shows the complete list of reserved keywords.

Table 1.4 List of Java Keywords

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

A Complex Example: `DrawFigures1`

The `println` statement can be used to draw text figures as output. Consider the following more complicated program example (notice that it uses two empty `println` statements to produce blank lines):

```
public class DrawFigures1 {
    public static void main(String[] args) {
        System.out.println("   /\\");
        System.out.println("  /  \\");
        System.out.println(" /    \\");
        System.out.println(" \\    /");
        System.out.println("  \\  /");
        System.out.println("   \\/");
        System.out.println();
        System.out.println(" \\    /");
        System.out.println("  \\  /");
        System.out.println("   \\/");
        System.out.println("   /\\");
        System.out.println("  /  \\");
        System.out.println(" /    \\");
        System.out.println();
        System.out.println("   /\\");
        System.out.println("  /  \\");
        System.out.println(" /    \\");
        System.out.println("+------+");
        System.out.println("|      |");
        System.out.println("|      |");
        System.out.println("+------+");
        System.out.println("|United|");
        System.out.println("|States|");
        System.out.println("+------+");
        System.out.println("|      |");
```

```
        System.out.println("|      |");
        System.out.println("+------+");
        System.out.println("   /\\");
        System.out.println("  /  \\");
        System.out.println(" /    \\");
    }
}
```

The following is the output the program generates. Notice that the program includes double backslash characters (\\), but the output has single backslash characters. This is an example of an escape sequence, as described previously.

```
   /\
  /  \
 /    \
 \    /
  \  /
   \/
 \    /
  \  /
   \/
   /\
  /  \
 /    \
   /\
  /  \
 /    \
+------+
|      |
|      |
+------+
|United|
|States|
+------+
|      |
|      |
+------+
   /\
  /  \
 /    \
```

Comments and Readability

Java is a free-format language. This means you can put in as many or as few spaces and blank lines as you like, as long as you put at least one space or other punctuation mark

between words. However, you should bear in mind that the layout of a program can enhance (or detract from) its readability. The following program is legal but hard to read:

```
1  public class Ugly{public static void main(String[] args)
2   {System.out.println("How short I am!");}}
```

Here are some simple rules to follow that will make your programs more readable:

- Put class and method headers on lines by themselves.
- Put no more than one statement on each line.
- Indent your program properly. When an opening brace appears, increase the indentation of the lines that follow it. When a closing brace appears, reduce the indentation. Indent statements inside curly braces by a consistent number of spaces (a common choice is four spaces per level of indentation).
- Use blank lines to separate parts of the program (e.g., methods).

Using these rules to rewrite the `Ugly` program yields the following code:

```
1  public class Ugly {
2      public static void main(String[] args) {
3          System.out.println("How short I am!");
4      }
5   }
```

Well-written Java programs can be quite readable, but often you will want to include some explanations that are not part of the program itself. You can annotate programs by putting notes called *comments* in them.

Comment

Text that programmers include in a program to explain their code. The compiler ignores comments.

There are two comment forms in Java. In the first form, you open the comment with a slash followed by an asterisk and you close it with an asterisk followed by a slash:

```
/* like this */
```

You must not put spaces between the slashes and the asterisks:

```
/ * this is bad * /
```

You can put almost any text you like, including multiple lines, inside the comment:

```
/* Thaddeus Martin
   Assignment #1
   Instructor: Professor Walingford
   Grader:     Bianca Montgomery      */
```

The only things you aren't allowed to put inside a comment are the comment end characters. The following code is not legal:

```
/* This comment has an asterisk/slash /*/ in it,
    which prematurely closes the comment. This is bad. */
```

Java also provides a second comment form for shorter, single-line comments. You can use two slashes in a row to indicate that the rest of the current line (everything to the right of the two slashes) is a comment. For example, you can put a comment after a statement:

```
System.out.println("You win!"); // Good job!
```

Or you can create a comment on its own line:

```
// give an introduction to the user
System.out.println("Welcome to the game of blackjack.");
System.out.println();
System.out.println("Let me explain the rules.");
```

You can even create blocks of single-line comments:

```
// Thaddeus Martin
// Assignment #1
// Instructor:  Professor Walingford
// Grader:      Bianca Montgomery
```

Some people prefer to use the first comment form for comments that span multiple lines but it is safer to use the second form because you don't have to remember to close the comment. It also makes the comment stand out more. This is another case in which, if your instructor does not tell you to use a particular comment style, you should decide for yourself which style you prefer and use it consistently.

Don't confuse comments with the text of `println` statements. The text of your comments will not be displayed as output when the program executes. The comments are there only to help readers examine and understand the program.

It is a good idea to include comments at the beginning of each class file to indicate what the class does. You might also want to include information about who you are, what course you are taking, your instructor and/or grader's name, the date, and so on. You should also comment each method to indicate what it does.

Commenting becomes more useful in larger and more complicated programs, as well as in programs that will be viewed or modified by more than one programmer. Clear comments are extremely helpful to explain to another person, or to yourself at a later time, what your program is doing and why it is doing it.

In addition to the two comment forms already discussed, Java supports a particular style of comments known as *Javadoc comments*. Their format is more complex, but they have the advantage that you can use a program to extract the comments to make HTML files suitable for reading with a web browser. Javadoc comments are useful in more advanced programming and are discussed in more detail in Appendix B.

1.3 Program Errors

In 1949, Maurice Wilkes, an early pioneer of computing, expressed a sentiment that still rings true today:

> As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

You also will have to face this reality as you learn to program. You're going to make mistakes, just like every other programmer in history, and you're going to need strategies for eliminating those mistakes. Fortunately, the computer itself can help you with some of the work.

There are three kinds of errors that you'll encounter as you write programs:

- *Syntax errors* occur when you misuse Java. They are the programming equivalent of bad grammar and are caught by the Java compiler.
- *Logic errors* occur when you write code that doesn't perform the task it is intended to perform.
- *Runtime errors* are logic errors that are so severe that Java stops your program from executing.

Syntax Errors

Human beings tend to be fairly forgiving about minor mistakes in speech. For example, the character Yoda would lose points for his unusual grammar in any writing class, but we still understand what he means.

The Java compiler will be far less forgiving. The compiler reports syntax errors as it attempts to translate your program from Java into bytecodes if your program breaks any of Java's grammar rules. For example, if you misplace a single semicolon in your program, you can send the compiler into a tailspin of confusion. The compiler may report several error messages, depending on what it thinks is wrong with your program.

A program that generates compilation errors cannot be executed. If you submit your program to the compiler and the compiler reports errors, you must fix the errors and resubmit the program. You will not be able to proceed until your program is free of compilation errors.

Some development environments, such as Eclipse, help you along the way by underlining syntax errors as you write your program. This makes it easy to spot exactly where errors occur.

It's possible for you to introduce an error before you even start writing your program, if you choose the wrong name for its file.

Common Programming Error

File Name Does Not Match Class Name

As mentioned earlier, Java requires that a program's class name and file name match. For example, a program that begins with `public class Hello` must be stored in a file called `Hello.java`.

If you use the wrong file name (for example, saving it as `WrongFileName.java`), you'll get an error message like this:

```
WrongFileName.java:1: error: class Hello is public,
    should be declared in a file named Hello.java
public class Hello {
       ^
1 error
```

The file name is just the first hurdle. A number of other errors may exist in your Java program. One of the most common syntax errors is to misspell a word. You may have punctuation errors, such as missing semicolons. It's also easy to forget an entire word, such as a required keyword.

The error messages the compiler gives may or may not be helpful. If you don't understand the content of the error message, look for the caret marker (^) below the line, which points at the position in the line where the compiler became confused. This can help you pinpoint the place where a required keyword might be missing.

Common Programming Error

Misspelled Words

Java (like most programming languages) is very picky about spelling. You need to spell each word correctly, including proper capitalization. Suppose, for example, that you were to replace the `println` statement in the "hello world" program with the following:

```
System.out.pruntln("Hello, world!");
```

When you try to compile this program, it will generate an error message similar to the following:

```
Hello.java:3: error: cannot find symbol
symbol : method pruntln(java.lang.String)
```

Continued on next page

Continued from previous page

```
location: variable out of type PrintStream
        System.out.pruntln("Hello, world!");
                  ^
1 error
```

The first line of this output indicates that the error occurs in the file `Hello.java` on line 3 and that the error is that the compiler cannot find a symbol. The second line indicates that the symbol it can't find is a method called `pruntln`. That's because there is no such method; the method is called `println`. The error message can take slightly different forms depending on what you have misspelled. For example, you might forget to capitalize the word `System`:

```
system.out.println("Hello, world!");
```

You will get the following error message:

```
Hello.java:3: error: package system does not exist
        system.out.println("Hello, world!");
              ^
1 error
```

Again, the first line indicates that the error occurs in line 3 of the file `Hello.java`. The error message is slightly different here, though, indicating that it can't find a package called `system`. The second and third lines of this error message include the original line of code with an arrow (caret) pointing to where the compiler got confused. The compiler errors are not always very clear, but if you pay attention to where the arrow is pointing, you'll have a pretty good sense of where the error occurs.

If you still can't figure out the error, try looking at the error's line number and comparing the contents of that line with similar lines in other programs. You can also ask someone else, such as an instructor or lab assistant, to examine your program.

Common Programming Error

Forgetting a Semicolon

All Java statements must end with semicolons, but it's easy to forget to put a semicolon at the end of a statement, as in the following program:

```
1  public class MissingSemicolon {
2      public static void main(String[] args) {
3          System.out.println("A rose by any other name")
```

Continued on next page

Continued from previous page

```
4           System.out.println("would smell as sweet");
5       }
6   }
```

In this case, the compiler produces output similar to the following:

```
MissingSemicolon.java:3: error: ';' expected
        System.out.println("A rose by any other name")
                                                      ^
1 error
```

Common Programming Error

Forgetting a Required Keyword

Another common syntax error is to forget a required keyword when you are typing your program, such as `static` or `class`. Double-check your programs against the examples in the textbook to make sure you haven't omitted an important keyword.

The compiler will give different error messages depending on which keyword is missing, but the messages can be hard to understand. For example, you might write a program called `Bug4` and forget the keyword `class` when writing its class header. In this case, the compiler will provide the following error message:

```
Bug4.java:1: error: class, interface, or enum expected
public Bug4 {
       ^
1 error
```

However, if you forget the keyword `void` when declaring the `main` method, the compiler generates a different error message:

```
Bug5.java:2: error: invalid method declaration; return type required
    public static main(String[] args) {
                   ^
1 error
```

Yet another common syntax error is to forget to close a string literal.

A good rule of thumb to follow is that the first error reported by the compiler is the most important one. The rest might be the result of that first error. Many programmers